Main-Memory Foreign Key Joins on Advanced Processors: Design and Re-evaluations for OLAP Workloads

Yansong Zhang[1,2] Yu Zhang[3] Xuan Zhou[4] Jiaheng Lu[5]
[1] MOE Key Laboratory of DEKE, Renmin University of China, Beijing, China
[2] School of Information, Renmin University of China, Beijing, China
[3] National Satellite Meteorological Center of China, Beijing, China
[4] School of Data Science and Engineering, East China Normal University, Shanghai, China
[5] Department of Computer Science, University of Helsinki, Finland
zhangys_ruc@hotmail.com, yuzhang@cma.gov.cn, zhou.xuan@outlook.com, jiahenglu@gmail.com

ABSTRACT
The hash join algorithm family is one of the leading techniques for equi-join performance evaluation. OLAP systems borrow this line of research to efficiently implement foreign key joins between dimension tables and big fact tables. From data warehouse schema and workload feature perspective, the hash join algorithm can be further simplified with multidimensional mapping, and the foreign key join algorithms can be evaluated from multiple perspectives instead of single performance perspective. In this paper, we introduce the surrogate key index oriented foreign key join as *schema-conscious* and *OLAP workload customized* design foreign key join to comprehensively evaluate how state-of-the-art join algorithms perform in OLAP workloads. Our experiments and analysis gave the following insights: (1) *customized* foreign key join algorithm for OLAP workload can make join performance step forward than general-purpose hash jo*ins;* (2) each join algorithm shows strong and weak performance regions dominated by the cache locality ratio of input_size / cache_size with a fine-grained micro join benchmark; (3) the simple *hardware-oblivious* shared hash table join outperforms complex *hardware-conscious* radix partitioning hash join in most benchmark cases; (4) the *customized* foreign key join algorithm with surrogate key index simplified the algorithm complexity for hardware accelerators and make it easy to be implemented for different hardware accelerators. Overall, we argue that improving join performance is a systematic work opposite to merely *hardware-conscious* algorithm optimizations, and the OLAP domain knowledge enables surrogate key index to be effective for foreign key joins in data warehousing workloads for both CPU and hardware accelerators.

1. INTRODUCTION
Recent years have witnessed the debate between *hardware-conscious* or *hardware-oblivious* hash join algorithms. *Hardware-conscious* join algorithm is to optimize join algorithm with comprehensive considerations on hardware characteristics to obtain the maximal performance gains and its underlying assumption is that the hardware must be carefully used to improve join performance. The representative *hardware-conscious* join algorithm is radix partitioning based hash join, in which cache and TLB are carefully used to make memory access efficient. On the other hand, *hardware-oblivious* join is designed with simple algorithm without much considerations on hardware characteristics, and the assumption is that hardware is good enough to automatically hide the memory access latencies. The representative *hardware-oblivious* join algorithm is the no-partitioning hash join, in which only the simple shared hash table is employed for parallel working threads without the concerns on specified hardware characteristics.

The purpose of this paper is not to simply determine whether *hardware-conscious* or *hardware-oblivious* join algorithms perform better than the others, but to discover the intrinsic performance pattern and to exploit the key insight to understand why and when one join algorithm outperforms the others. To handle the real-world workloads, comprehensive benchmark evaluations are considered, instead of limited user defined workloads for performance testing. We argue that the best join algorithm may not be the fastest one, and the implementation complexity, the intermediate memory consumption, the heterogeneous platform adaptiveness and the compatibility with query engine are all important and indispensable dimensions for holistic consideration.

In particular, we review the state-of-the-art main-memory join algorithms and their limitations as follows.

First, the *hardware-oblivious* join algorithm [1] uses the shared hash table for building and probing without much considerations for how to eliminate cache conflicts. In contrast, the *hardware-conscious* join algorithm [2] uses radix partitioning [3] method and many hardware tuning configurations to divide big join table into cache fit small partitions to improve cache locality. With the partitioning space cost, *hardware-conscious* join algorithm always outperforms the *hardware-oblivious* join algorithm [18]. These approaches are beneficial for OLAP workloads for the foreign key join between dimension tables and big fact tables dominating the OLAP performance. But the conclusions need re-evaluations based on OLAP schema, workloads, update mechanism features, and the OLAP domain knowledge can also further improve foreign key join performance with *customized* structure and algorithm.

Second, comprehensive join benchmarks, rather than the isolated user-defined workloads should be considered for performance evaluations. The recent work [18] extended the workloads for more scenarios. Their work focused on join evaluations with a small relation (smaller than probe relation) and a big relation (with the same size of probe relation). This experimental design has two major limitations. (1) *hardware-oblivious* join algorithm is sensitive to the cache locality ratio of *input_size*/*cache_size* instead of the size of join relation R or the ratio of |R|/|S|. To describe the join performance, we should vary the join table size according to L1, L2, L3 cache slice, and LLC size to measure the join performance pattern. (2) database benchmarks are widely used for

the academic and industry performance evaluations with different schemas, e.g., SSB for star schema, TPC-H for snow-flake schema and TPC-DS for snow storm schema, and the joins in these benchmarks represent the big fact table joining with different size of dimension tables. The benchmark tests can evaluate how different join algorithms perform in real-world workloads with different size of tables.

Third, hardware accelerators come to be main-stream high performance computing platforms, the join algorithm designs and optimizations should consider the heterogeneous platform adaptiveness to avoid tightly coupled optimizations with CPU cache hierarchy. The cache-centric optimizations focus on cache locality in private cache (L1, L2, L3 cache slice), while private cache in GPU and Xeon Phi accelerators is commonly small, so that it is to make join algorithm adaptive to be performed in parallel. One feasible solution is to extend *hardware-oblivious* join algorithm to *platform-oblivious* join algorithm i.e., simplifying hash table to reduce branching, using one to one map to reduce conflicts in building phase and using shared probing for massive parallel working threads.

In this paper, we combine *schema-conscious* and *OLAP workload customized* design with join algorithm through heterogeneous platform perspective, benchmark evaluation perspective and systematic optimization perspective. The contributions of this paper are summarized as follows:

- **Schema-conscious and OLAP workload customized design for join**. We introduce surrogate key index mechanism to relational data warehouse. The column-wise index simplifies foreign key join as array index referencing (AIR) by creating surrogate key index for PK-FK constraint tables. Surrogate key index enables relational database to perform a multidimensional style operation like [12], and the surrogate key index mechanism makes array join in [18] to be an *OLAP workload customized* design for relational data warehouses.
- **Benchmark based evaluations**. We start with a *cache-conscious* join benchmark with fine-granularity tests, where the probe table remains fixed-length and the join table varies its size to different proportions of L1, L2, L3 cache size. Our fine-granularity experimental results give a performance curve chart for each join algorithm to discover the performance strength and weakness regions and the dependency between join table size and cache size. We also employ SSB, TPC-H, TPC-DS benchmarks to exploit how these join algorithms perform for real-world workloads, and our analysis on the comprehensive experimental results shed the lights on query optimizer design in practice.
- **Platform evaluations**. The emerging trend is to accelerate the costly relational join operation with new hardware based on more cores and simultaneous massive-threading mechanism such as GPU[4][5], APU[10], Phi[6], and FPGA[8]. We implemented the join algorithm for Xeon Phi and NVIDIA K80 GPU platforms, and designed the experiments to evaluate the join performance for different processor architectures. Our experimental results discovered how caching and simultaneous massive-threading mechanisms dominate the join performance in different workloads and platforms. We also found that memory efficiency is another key consideration for joins to maximize co-processor's on-board device memory utilization rate.

The rest of this paper is organized as follows: Section 2 introduces the background of representative in-memory join algorithms. The surrogate key index mechanism is described in Section 3. Section 4 presents the experimental evaluations. Section 5 analyzes the related work, and Section 6 concludes the paper.

2. PRELIMINARIES

The essential difference between *hardware-oblivious* and *hardware-conscious* join algorithms is whether hardware is good enough to optimize memory access latencies. The main roadmaps to reduce memory access latency can be classified into two categories: (1) caching mechanism and (2) simultaneous multi-threading approach. The effectiveness of caching mechanism relies on the LLC size, x86 processors are commonly designed with small LLC size (2.5MB*#core), while the latest KNL processor can configure at most 16 GB on-board high bandwidth memory as LLC, the 6th-generation Skylake also supports large memory-side cache [20], the trend of increasing LLC size enables *hardware-oblivious* join algorithm to be used for more workloads. The simultaneous multi-threading mechanism overlaps memory access latency with massive threads, the Xeon Phi's hyper threading and GPU's SIMT technique support hundreds or thousands of simultaneous threads to overlap memory access latency. From hardware feature perspective, *hardware-conscious* join algorithm is majorly designed for multiple cache hierarchy of x86 processors, *hardware-oblivious* join algorithm is adaptive to hardware accelerator's architecture with massive hardware simultaneous threads. In this paper, we shall focus on *hardware-oblivious* join algorithm on hardware accelerator platforms.

Another important perspective is combining *schema-conscious* and *OLAP workload customized* design into join algorithm optimizations. For data warehouse schemas, as shown in Figure 1, the multidimensional dataset can be modeled as data cube, the fact data can be located by multiple dimensions. Relational model organizes the data as dimension tables and fact table with PK-FK referencing constraints between them, missing the features that dimension tables can be mapped to dimensions and fact data can directly map to corresponding dimension items. Thus, the foreign key join in OLAP workloads can be customized by using dimension table as dimension and transforming hash join as FK mapping to dimension, simplifying the hash probing as address probing.

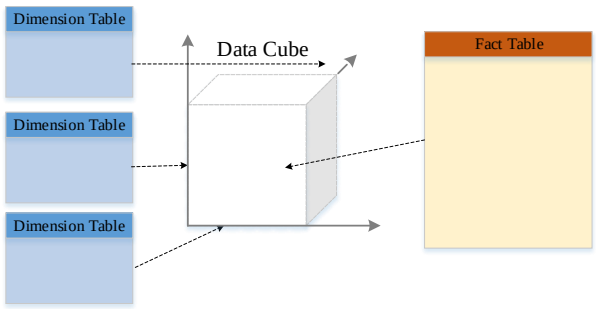


Figure 1. Multi-dimensional model and relational model.

In this paper, we majorly evaluated three representative join algorithms: *hardware-oblivious* no partitioning hash join, *schema-conscious* and *OLAP workload customized* AIR, and *hardware-conscious* radix partitioning hash join. Three join algorithms are designed with two important perspectives: hardware feature and database systematic design.

2.1 No Partitioning Join

Let *R* denote the dimension table with primary key, and *S* the fact table with foreign key. The join between *R* and *S* is to locate the corresponding tuple in *R* with the same key of tuple in *S*. NPO algorithm from [1] is a no-partitioning join algorithm, which builds a shared hash table from *R*. For multiple-thread parallel processing, NPO algorithm divides the input relations *R* and *S* into equi-sized portions that assigned to working threads. As shown in Figure 2, during *build* phase, the hash table is shared among all the working threads. To enable the concurrent insertions on hash table, each bucket is protected via a *latch* for a thread to obtain before inserting a tuple. [2] has optimized the hash table by combining the lock and hash bucket together to reduce cache misses. For the big input relation, the *build* phase is very costly.

NPO algorithm does not consider hardware parameters such as cache size or TLB entries, and the performance is majorly affected by table size. When shared hash table from *R* is small enough to be held in cache, NPO is efficient; but when shared hash table size exceeds cache size, NPO suffers from cache miss latencies. Modern processors commonly use *auto-prefetching*, *out-of-order execution* and *simultaneous multi-threading* mechanisms to overlap or hide cache miss latencies. For CPU platform, the hash probing overhead is large for limited cores and threads. The hardware accelerators like Phi and GPU strengthen the *simultaneous multi-threading* feature with hundreds or thousands of threads, the hash probing performance of NPO can be accelerated by them.

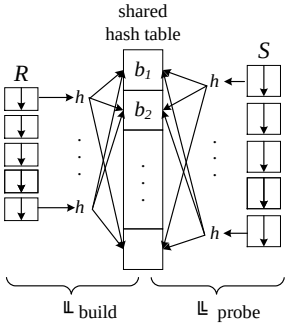
2.2 AIR algorithm

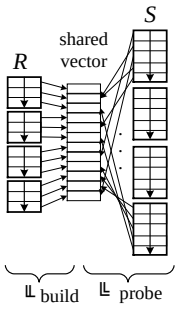
AIR algorithm is an *array-store* oriented foreign key join algorithm. As shown in Figure 3, relation *R* is stored as array, the array index is used as primary key of *R*, and the foreign key of *S* is the array index of referenced tuple of *R*, the foreign key join between *R* and *S* is simplified as array addressing on *R*. This feature is also used in CAT[17] and array join[18] with PRIMARY KEY AUTOINCREMENT constraint and dense primary key. [12] further optimized the shared hash table with a shared vector, each tuple of *R* is mapping to a unique vector cell according to primary key, the hash probing is also simplified as directly accessing vector cell by mapping foreign key value to vector index.

Compared with NPO algorithm, the vector is simpler than hash table which contains *next* pointer for overflow bucket and cannot be directly implemented on GPU platform. If all the tuples in table *R* is selected, hash table size is larger than vector for storing key, value and the additional meta data such as *head* pointer, *next* pointer; in *build* phase, the tuple is mapped to unique cell in vector without conflict control opposite to *latch* mechanism in building shared hash table of NPO; the complexity of hash probing can be expected as $O(1)$, but hash probing involves many cycles on computing hash functions, loading buckets, matching key and probing in overflow bucket etc., which need to be further optimized with SIMD on Phi platform[6][7], while AIR algorithm directly uses key for address probing without storing key/hashing key and matching key, each key is strictly mapped to unique cell in vector without further computing cycle consumption.

AIR uses the fixed length vector for address probing, each tuple is mapped to the unique cell in vector without conflicts, while NPO uses the filtered tuples to build shared hash table with *latch* mechanism. For query with high selectivity, the shared vector is smaller than shared hash table, for query with low selectivity, the shared vector may be larger than shared hash table. In OLAP benchmarks, queries usually have high selectivity on single dimension table in roll-up and drill-down operations, the shared vector is commonly smaller than shared hash table; with compression on dimension tables, the shared vector can be even smaller. Shared vector is adaptive to data warehouse workloads, on one hand, the dimension tables in data warehouses are commonly small size and increasing slowly, the shared vector for dimension table is usually small size; on another hand, nowadays processors have large LLC size for caching shared vector, e.g., Xeon E7 8890 v4's 60 MB L3 cache can hold a shared vector(*int*_8) with 60,000,000 rows, the fixed length shared vector is efficient for processors and data warehouse workloads. Considering the large LLC size, small compact vector size and small dimension size, AIR algorithm is customized for OLAP workloads.

NPO algorithm is *hardware-oblivious* for multicore CPU platform, but how to design efficient hash join is a complex issue [11]. For hardware accelerator platforms such as GPU, Phi or FPGA, the complex hash table, hashing schema, hash function, SIMD





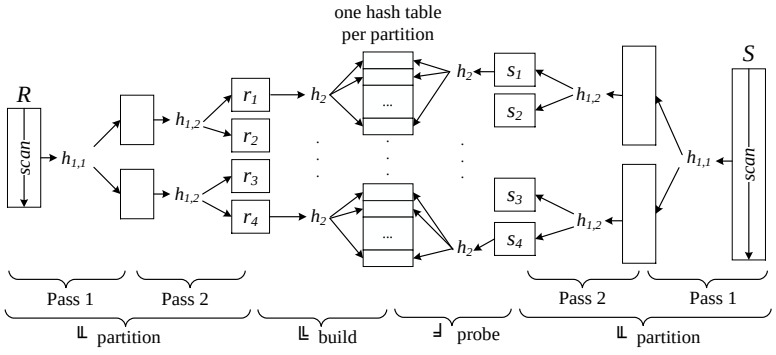


Figure 2. No partitioning join **Figure 3. AIR algorithm**

Figure 4. Radix join

optimizations etc. need to be re-designed with specified platform features[4][7][8][10]. NPO algorithm is *hardware-oblivious* but not *platform-oblivious* join approach. AIR algorithm is even simpler than NPO algorithm in data structure, *build* phase and *probe* phase, the vector structure is compatible for different hardware platforms, the address probing operation involves less computing optimization requirements. AIR is a *customized* join algorithm for data warehousing workloads, the *customized* design with simple structure and algorithm also enable AIR to be *platform-oblivious* for heterogonous platforms.

The OLAP benchmarks, e.g., SSB, TPC-H and TPC-DS use surrogate key (consecutive values without semantic information) as primary key, Table 1 illustrates how to map surrogate key to vector index. [12] proposed array store to guarantee directly mapping primary key to vector index, it is not a OLTP workload design but a OLAP workload design for the read-only feature of data warehouses. The new trend of analytical in-memory database is to combine OLTP and OLAP processing inside single engine, the representative systems are Hyper [13] and SAP HANA [15]. The *copy-on-write*, MVCC, insert-only mechanisms for OLTP conflict with strictly position constraint of array store, we need a relaxed mechanism to map out-of-order keys to vector addresses, and how to maintain array index with update workload.

2.3 Parallel Radix Join

We use PRO algorithm from [2] as parallel radix partitioning join algorithm. PRO majorly optimizes the *build* and *probe* phases with hardware parameters to tune the performance. In *partition* phase, the major latency is caused by TLB miss. TLB caches the virtual memory mapping entries, the number of TLB entries are defined by hardware, if the number of created partitions exceeds the number of TLB entries the *partition* phase may cause TLB misses. The radix partitioning optimizes TLB misses by partitioning both input relations *R* and *S* in multiple passes, as shown in Figure 4, the first pass looks at a different set of bits from hash function $h_{1,1}$, the second pass looks at the other set of bits from hash function $h_{1,2}$, in 2^{nd} pass, all partitions produced by 1^{st} pass guarantee the partitioning fan-out never exceeds the hardware limit given by the number of TLB entries. For typical data sizes, two or three passes are sufficient to create cache-sized partitions without TLB misses. Hash tables are built over each cache-sized partitions of table *R*. In *probe* phase, all s_i partitions are scanned and probe the respective r_i partitions for join matches.

Two important hardware parameters of radix join are the maximum fanout per radix pass and partition size which are defined by the number of TLB entries and cache size. [2] also showed that radix join can also works well with misconfiguration of either parameters.

PRO algorithm optimizes parallel radix join in *partition* phase to avoid thread contention in creating a shared set of partitions. Each thread scans the *R* and *S* relations twice, the first scan computes a set of histograms over the input data for exact output size for each thread and partition, by computing a prefix-sum over the histogram, a contiguous memory space is allocated for the output and each thread pre-computes the exclusive location for its output. With these optimizations, all threads can perform parallel hash join on each partition without needs to synchronize.

PRO is designed for cache-centric architecture, the hypothesis is that caching is superior to simultaneous multi-threading mechanism, so that the big relations are divided into cache-fit small partitions to guarantee in-cache hashing. That is right for x86 processors for the limited threads and large LLC, but it is not suitable for accelerators with massive threads and small cache size (e.g., NVIDIA Tesla K80 has 4992 cuda cores and 1.5MB L2 cache). PRO algorithm doubled the memory space in *partition* phase, the performance gain is preferred for multicore CPU platform with large memory. But for the emerging accelerator platforms like GPU and Phi, the on-board memory size is limited and expensive, PRO algorithm sacrifices the memory efficiency.

Table 1. Surrogate key in benchmarks

	Table	Surrogate Key	Key-address map function
SSB	customer	1,2,3,…	*f(key)=key-1*
	supplier	1,2,3,…	*f(key)=key-1*
	part	1,2,3,…	*f(key)=key-1*
	date	19920101, 19920102, …	*f(key)= date(key)-date(key_0)*
TPC-H	customer	1,2,3,…	*f(key)=key-1*
	supplier	1,2,3,…	*f(key)=key-1*
	part	1,2,3,…	*f(key)=key-1*
	nation	0,1,2,…	*f(key)=key*
	region	0,1,2,…	*f(key)=key*
TPC-DS	call_center	1,2,3,…	*f(key)=key-1*
	catalog_page	1,2,3,…	*f(key)=key-1*
	customer	1,2,3,…	*f(key)=key-1*
	customer_address	1,2,3,…	*f(key)=key-1*
	customer_demographics	1,2,3,…	*f(key)=key-1*
	date_dim	2415022, 2415023, …	*f(key)= key-key_0*
	household_demographics	1,2,3,…	*f(key)=key-1*
	income_band	1,2,3,…	*f(key)=key-1*
	item	1,2,3,…	*f(key)=key-1*
	promotion	1,2,3,…	*f(key)=key-1*
	reason	1,2,3,…	*f(key)=key-1*
	ship_mode	1,2,3,…	*f(key)=key-1*
	store	1,2,3,…	*f(key)=key-1*
	time_dim	0,1,2,…	*f(key)=key*
	warehouse	1,2,3,…	*f(key)=key-1*
	web_page	1,2,3,…	*f(key)=key-1*
	web_site	1,2,3,…	*f(key)=key-1*

3. SURROGATE KEY INDEX

Surrogate key is widely used in data warehouses, surrogate key uses the simple consecutive integer sequence as primary key, and can also be produced with PRIMARY KEY AUTOINCREMENT constraint in databases.

For an in-memory column store, a surrogate key can be mapped to the offset address of dimension tuple. From the perspective of multidimensional model, the surrogate key can be mapped to dimension coordinate axis to identify the fact data in data cube. From the perspective of the relational model, the surrogate foreign key can be used as join index to map foreign key of a fact tuple to the offset address of dimensional tuple. The surrogate key represents the key-address mapping for column store.

3.1 Creating Surrogate Key Index

In data warehouse schemas, dimension tables commonly use surrogate key as primary key e.g., in benchmarks of SSB, TPC-H and TPC-DS. When tables with PK-FK referencing constraints do not use surrogate key, we can create surrogate key index on them for accelerating foreign key join performance. The surrogate key index consists of two parts, one is surrogate key index as new primary key column with incremental integer values, the other is surrogate foreign key index as new foreign key column with updated value from surrogate key index column.

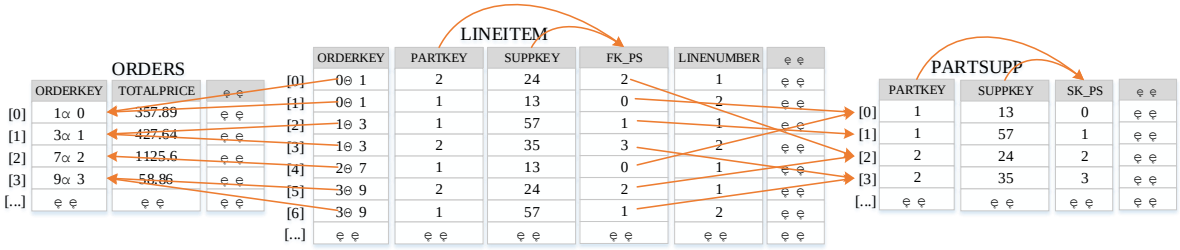


Figure 5. Creating surrogate key and updating foreign key as surrogate foreign key.

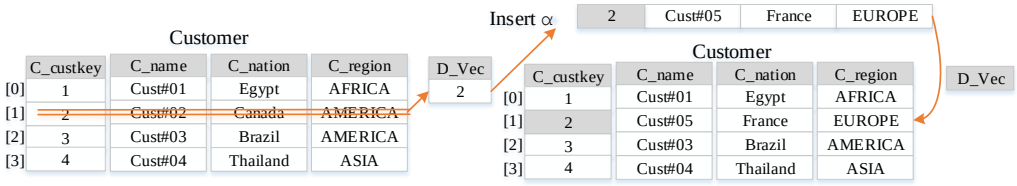


Figure 6. Delete vector and surrogate key reuse mechanism.

In snow-flake or snow-storm schema, there are multiple fact tables with PK–FK references without surrogate key mechanism. For example, in TPC-H, the primary key of *lineitem* is (*orderkey*, *linenumber*), the primary key of *orders* table is (*orderkey*), the primary key of *partsupp* table is (*partkey*, *suppkey*). *lineitem* references *orders* table by *orderkey* column with the same order because both *lineitem* and *orders* tables are cluster indexed with *orderkey* column. And *lineitem* references *partsupp* table with composite key (*partkey*, *suppkey*). The *orderkey* in *orders* table is not surrogate key with inconsecutive values, and *partsupp* table has no surrogate key at all. As fact tables store historical data with read-only access, we can implement surrogate key mechanism on fact tables for surrogate key referencing. In Figure 5, as primary key of *orders* table is first partial key of *lineitem* table, a merge join can be performed on *orderkey* of *lineitem* table and *oderkey* column of *orders* table, the *orderkey* in *orders* table is updated with column offset addresses (array index) to transform *orderkey* column into surrogate key, at the same time *orderkey* column in *lineitem* table is updated with the same value accordingly. For *partsupp* table, we need to add an additional surrogate key index as column *SK_PS* in *partsupp* table and an additional surrogate foreign key index as column *FK_PS* in *lineitem* table, an extra join between *lineitem* table and *partsupp* table is invoked at idle time to update surrogate foreign key column *FK_PS* so as to enable surrogate key address probing. The surrogate key update can be incremental, the join between *partsupp* and *lineitem* can be divided into two parts, conventional join for new inserted tuples and surrogate key referencing for tuples with updated surrogate key.

In TPC-DS, for example, the composite primary key of *store_sales* table is (*ss_item_sk*, *ss_ticket_number*), the composite key and foreign key of *store_return* table is (*sr_item_sk*, *sr_ticket_number*), we can also add additional surrogate key column in both *store_sales* table and *store_return* table to enable surrogate key referencing with similar update mechanism.

3.2 Updates on Surrogate Key

When surrogate key is used to store dimension coordinates, we need to keep the surrogate key consecutive. This is an additional constraint for update operations.

For insertion operation, new tuples are usually appended to the table. Each new surrogate key is allocated by invoking *max*()+1.

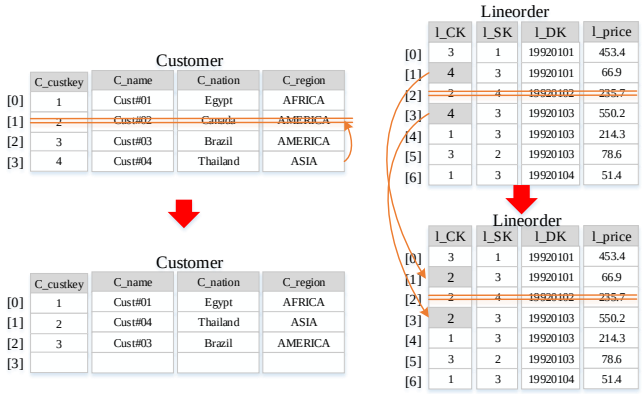


Figure 7. Consolidation mechanism of dimension table.

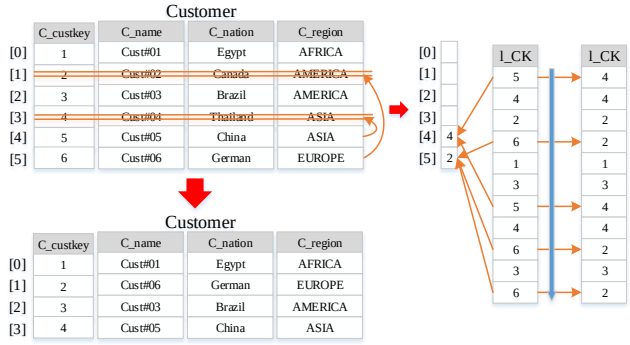


Figure 8. Batched consolidation of dimension table.

For deletion operation, most in-memory databases, e.g., MonetDB and Vectorwise, commonly adopt a lazy deletion mechanism, in which a deletion vector is used to mark the positions of deleted tuples instead of physically removing the tuples. Deletion leaves holes in a relational table, which can be re-assigned to newly inserted tuples. This keep surrogate key always consistent with

surregate key	d_datekey	d_date	d_dayofweek	d_month	d_year
1	19920101	1-Jan-92	Thursday	January	1992
2	19920102	2-Jan-92	Friday	January	1992
3	19920103	3-Jan-92	Saturday	January	1992
4	19920104	4-Jan-92	Sunday	January	1992
5	19920105	5-Jan-92	Monday	January	1992
6	19920106	6-Jan-92	Tuesday	January	1992
7	19920107	7-Jan-92	Wednesday	January	1992
8	19920108	8-Jan-92	Thursday	January	1992
9	19920109	9-Jan-92	Friday	January	1992
10	19920110	10-Jan-92	Saturday	January	1992
11	19920111	11-Jan-92	Sunday	January	1992
12	19920112	12-Jan-92	Monday	January	1992
13	19920113	13-Jan-92	Tuesday	January	1992
14	19920114	14-Jan-92	Wednesday	January	1992
15	19920115	15-Jan-92	Thursday	January	1992

surregate key	d_datekey	d_date	d_dayofweek	d_month	d_year
2	19920102	2-Jan-92	Friday	January	1992
9	19920109	9-Jan-92	Friday	January	1992
5	19920105	5-Jan-92	Monday	January	1992
12	19920112	12-Jan-92	Monday	January	1992
3	19920103	3-Jan-92	Saturday	January	1992
10	19920110	10-Jan-92	Saturday	January	1992
4	19920104	4-Jan-92	Sunday	January	1992
11	19920111	11-Jan-92	Sunday	January	1992
1	19920101	1-Jan-92	Thursday	January	1992
8	19920108	8-Jan-92	Thursday	January	1992
15	19920115	15-Jan-92	Thursday	January	1992
6	19920106	6-Jan-92	Tuesday	January	1992
13	19920113	13-Jan-92	Tuesday	January	1992
7	19920107	7-Jan-92	Wednesday	January	1992
14	19920114	14-Jan-92	Wednesday	January	1992

Dim_vec	
[1]	Thursday
[2]	Friday
[3]	Saturday
[4]	Sunday
[5]	Monday
[6]	Thursday
[7]	Wednesday
[8]	Thursday
[9]	Friday
[10]	Saturday
[11]	Sunday
[12]	Monday
[13]	Thursday
[14]	Wednesday
[15]	Thursday

Figure 9. Logical surrogate key index.

dimension coordinates. If the dimension is small or deletion produces less holes, we can simply leave the holes in surrogate key index, otherwise, we have to re-organize the surrogate key.

Figure 6 illustrates the rationale of the surrogate key deletion and reuse mechanism. Sometimes, when deletion leave too many holes in the table, we can perform consolidation, which compresses the table space and reassigns the surrogate keys. As an example in Figure 7, when tuple with surrogate key *C_custkey*=2 is deleted, the fact tuples which reference this tuple are also deleted. We can move the last tuple (*C_custkey*=4) to current tuple slot. Accordingly, the surrogate foreign key column *l_CK* needs an update to assign *l_CK*=2 where original value is 4. Such a consolidation process can be done in a batch, as shown in Figure 8. When dimension tuples with *C_custkey*=2, *C_custkey*=4 are deleted, we move the last two tuples in tail to fill the deleted tuples, and the *C_custkey* is refreshed from 5, 6 to 4, 2. We generate a deletion vector to record the tuple movements. A NULL cell in the vector represents no change on the corresponding tuple, and a non-empty cell represents that the original tuple's surrogate key is changed to the value in the cell. Based on the deletion vector, the foreign key column can be updated accordingly. Consolidation is not compulsory, as it is usually at the cost of a full foreign key join. A system can selectively perform consolidation at non-peak hours.

For update operation, as a surrogate key does not contain functional information, it is usually kept intact. In-place update is usually performed on the other attributes.

3.3 Logical Surrogate Key Index

Most analytical databases support insert-only mode to simplify update mechanism. Such an update operation is always divided into two operations: insertion of the new tuple with the update and deletion of the original tuple. The constraint of surrogate key index with surrogate key value as offset address may incur significant overheads to insert-only update. Thus, we propose a relaxed surrogate key index, named logical surrogate key index.

In a logical surrogate key index, the surrogate key no longer represents offset address but a logical sequence. The only constraint is to use consecutive sequence as surrogate key. The logical surrogate key deletion vector reuse mechanism is also available. The projection operation produces a surrogate vector as shown in Figure 9, the logical surrogate key value is now used as offset address in surrogate vector, the projected attribute value can be directly mapped to surrogate vector cell.

The surrogate vector is a switcher between logical surrogate key enabled table and physical surrogate key enabled table. Moreover, if we employ a main-memory database engine as dimension table management engine without in-place update mechanism, we can still achieve the surrogate key mapping mechanism by adding an API to project attributes to a physical surrogate vector according to logical surrogate key value as vector index.

4. Surrogate Key Vector Referencing

Vector oriented processing is widely adopted by analytical main memory databases. It can be classified into three types: SIMD vector processing, vectorized processing, and vector referencing.

Modern processors are equipped with wide register, for example, Intel Haswell supports 256-bit SIMD instructions to speedup packed data processing. GPGPU and Xeon Phi coprocessors support 512-bit SIMD instructions to further speedup data processing efficiency with a single instruction. SIMD is a hot research topic in query processing. The investigated operators include sorting, hash key calculating and hash probing. SIMD provides a register level vector processing support with dense packed data layout. As shown in Figure 10, SIMD instructions can calculate multiple hash map values in parallel.

Vectorized processing is the fundamental feature of state-of-the-art main-memory databases. Its major principle is to operate on the granularity according to the size of L1 cache. In other words, each vector is a query processing unit that should fit in the L1 cache, minimizing the materialization cost and boosting the performance of query processing significantly.

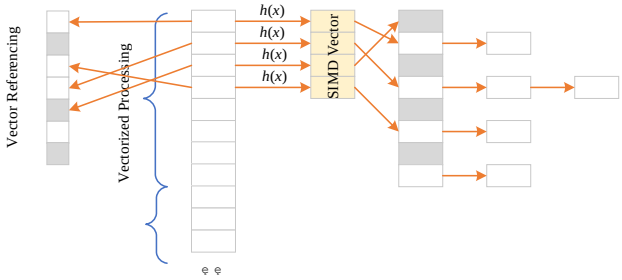


Figure 10. SIMD processing vs. vectorized processing vs. vector referencing.

Our surrogate key index based AIR algorithm can be regarded as vector referencing approach. Surrogate key index is normally implemented as a vector or a bitmap. The foreign key join is

performed as referencing operation on the vector, which translates foreign key values into the offset addresses of vector. Vector referencing and the SIMD based vectorized processing techniques can be used together, to achieve higher performance.

3.1 Vector referencing

Hash table is the key optimization technique for main-memory database query processing. A hash probing operation involves many CPU instructions, including those for hash probing, hash key matching, linear search, overflow bucket search, etc. Many CPU cycles are consumed in hash operation.

With a surrogate key index, each fact tuple has the address information of referenced dimension tuples from surrogate foreign key attributes, and can directly access corresponding dimensional tuple. The overhead of building hash table and hash probing is reduced. As illustrated by the example in Figure 11, *l_CK* and *l_SK* are surrogate foreign key, and *l_DK* can be mapped as surrogate foreign key; the dimension columns are used as surrogate vectors to be directly referenced by fact tuples. The foreign key of fact tuple can be considered as a native join index[22].

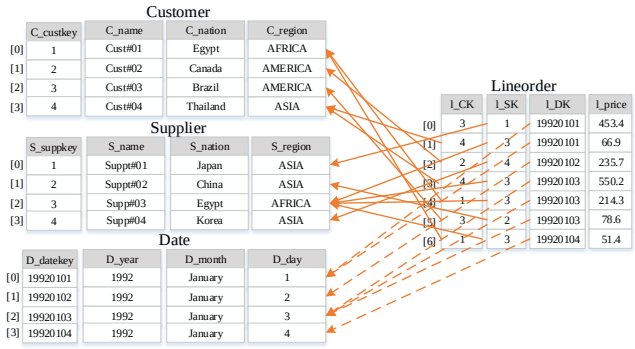


Figure 11. Surrogate key referencing.

For OLAP queries with multiple or complex predicate expressions on dimension tables, we can first process the predicates on the dimension tables and filter out dimension tuples that should not participate in the joins. This turns each dimension table into a bitmap, which can then be joined with the fact table using the surrogate key index. Similar bloom filter technique is adopted by databases such as Oracle Smart Scan, Vectorwise, etc.

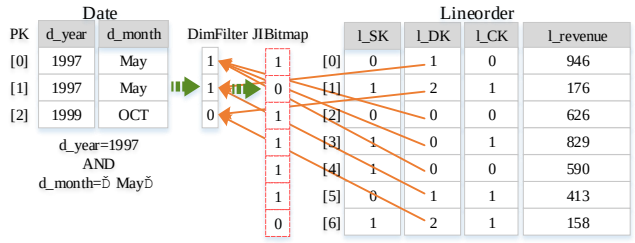


Figure 12. Surrogate bitmap referencing.

Figure 12 illustrates the example of surrogate bitmap referencing. Predicates on small dimension table are performed to generate a bitmap (DimFilter in Figure 12) to identify which tuple satisfies the predicate expressions. Supported by surrogate key mechanism, the foreign key column can directly refer to the surrogate bitmap to accomplish the join. A bitmap for the fact table can be materialized to record join filtering result and used as bitmap join index (JIBitmap in Figure 12) to filter other foreign key columns for the following surrogate bitmap referencing operations on other dimension tables. As the surrogate bitmap is always very small, the filtering is more efficient and accurate than bloom filter.

OLAP query can be modeled as SPJGA (select, project, join, group, aggregate) operation. The projection attributes are attributes in GROUP BY clause. The general-purpose surrogate vector referencing is illustrated in Figure 13. The predicate *c_region*='AMERICA' is executed, the filtered GROUP BY attribute *c_natio*n is projected as surrogate vector. OLAP query commonly uses low cardinality attributes as hierarchy attributes, the grouping attributes can be stored as *integer* type under light-weight dictionary encoding technique. Furthermore, we can dynamically compress the projected vector to assign shorter code for surrogate vector like dotted box in Figure 13. We have applied this dynamically dictionary compression technique in [12].

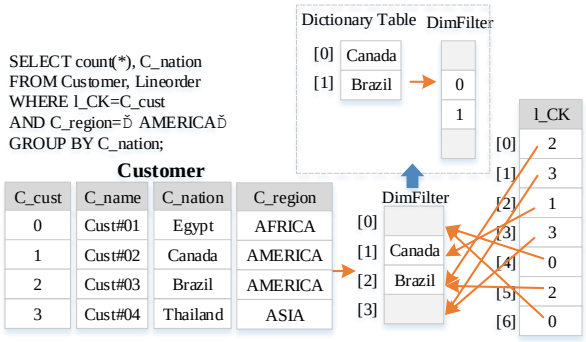


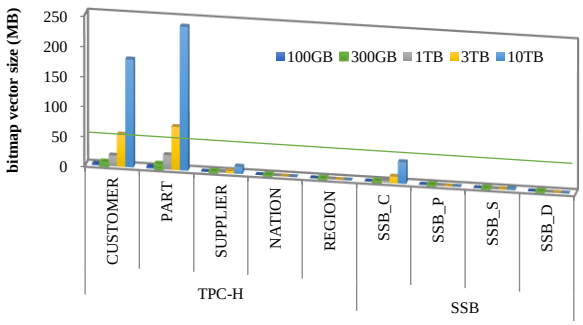
Figure 13. Surrogate vector referencing.

For typical OLAP queries, the number of groups in a single dimension table is usually small, so that the result is small enough for drill-down or roll-up operations. Therefore, in our experiments, we use *int*_8, *int*_16, *int*_32 as the vector width to measure the performance of surrogate vector referencing.

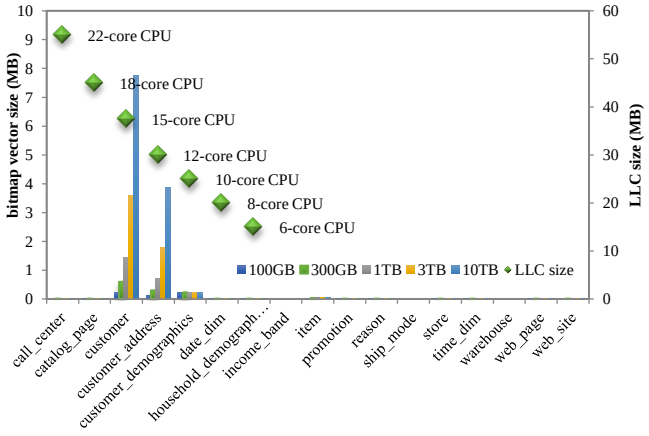
3.2 Schema-aware vector referencing mechanism

SIMD based and vectorized query processing use the size of register or L1 cache size to define the length of vectors. In contrast, the vector length in AIR algorithm depends on the size of the dataset. In turn, the length of vector affects the cache efficiency of vector referencing.

Figure 14 gives a statistic chart about how the size of surrogate bitmap affects the performance on typical benchmarks of SSB, TPC-H and TPC-DS (SF=100, 300, 1000, 3000, 10000). In Figure 14 (a), the line represents the 22-core CPU's LLC size (Xeon E5-2600 v4, 55MB LLC). SSB is a denormalized schema of TPC-H, and the dimension table size is modified to match real-world businesses. The surrogate bitmap vectors of dimension tables are always very small, even for dataset of SF=10000, the biggest surrogate bitmap vector is smaller than the latest CPU's LLC size. TPC-H has two big dimension tables *customer* and *part*, the surrogate bitmap size is smaller than 55 MB with datasets of SF=100, 300 and 1000, for bigger datasets of SF=3000 and 10000, the surrogate bitmap size is larger than 55 MB.



(a) TPC-H and SSB



(b) TPC-DS

Figure 14. Dimension bitmap sizes for dimension tables in different dataset sizes (SF=100,300,1000,3000 and 10000).

TPC-DS's schema is more complex than TPC-H and SSB, and contains more dimension tables. However, its dimension tables are typical slowly increasing dimensions, and the corresponding surrogate bitmap size is much smaller. In Figure 14 (b), the left Y axis represents the size of bitmap vector size(MB), the right Y axis represents the LLC size of different types of CPUs. The biggest surrogate bitmap size is 7.75 MB, even smaller than low-end 6-core CPU's LLC size.

As most surrogate bitmaps are cache fit, we can use the fixed length surrogate vector for different OLAP queries to simplify join algorithm design with high performance.

3.3 AIR implementations on accelerators

Like NPO algorithm, AIR uses the shared vector for foreign key referencing. In building phase, relation *R* is parallel scanned by logical partition with working thread, the filtered values (bitmap or compact value) are mapped to cells in surrogate vector by surrogate key value. In probing phase, relation *S* is also logically partitioned by thread number for parallel vector referencing operations.

Xeon Phi has similar cache architecture as multicore CPU, the code can be compiled with icc for Phi version. The more threads (4 thread for one core) and less cache hierarchy (has no L3 cache) make the performance of AIR on Phi different from AIR on CPU.

In this paper, we only use shared memory of GPU for join result counter, the surrogate vector is accessed from global memory. The parallelism is configured with BLOCK_NUM and THREAD_NUMBER parameters for maximal performance.

The implementations of AIR on different accelerator platforms are similar to each other without much hardware specialized optimizations, the simple vector structure and vector addressing operation makes AIR easy to be implemented on different accelerator platforms.

As a summary, AIR is designed with systematic optimizations. From schema perspective, AIR uses the multidimensional model to simplify equi-join as dimension mapping. From workload perspective, the append-only mode update, small and slow increasing dimension guarantee the surrogate vector to be small and efficient. From index perspective, surrogate key index is low cost in space and maintenance for OLAP workloads. From performance perspective, the vector referencing mechanism of surrogate key index reduces the CPU cycle consumptions for key oriented hash probing. Finally, AIR is simple enough to reduce implementation overhead for heterogenous accelerator platforms.

5. EXPERIMENTAL EVALUATIONS

This paper re-focused on join performance evaluation with three new perspectives, introducing a *schema-conscious* and *OLAP workload customized* design join algorithm AIR, evaluating join performance on industry benchmarks and illustrating how hardware features dominate the performance of *platform-oblivious* join algorithm in two representative accelerator platforms.

The experiments are designed with 3 parts. Part 1 is to evaluate the dependency between cache locality ratio of *input_size*/*cache_size* and performance for three join algorithms. [1] and [2] all focused on data warehouse workloads, they only choose two workloads as small table join and big table join to demonstrate whether *hardware-oblivious* algorithm or *hardware-conscious* algorithm performs better. But we still need further evaluations to discover why one join algorithm outperforms others and exploit how to use the conclusions for query optimizer design.

Opposite to the related approaches, we design a *cache-conscious* join benchmark to evaluate join performance. In the CCJB (*Cache-Conscious* Join Benchmark), a group of join workloads are designed to measure the join performance pattern, the input (shared hash table or shared vector) size is configured based on the proportion of cache size to exploit how join performance is influenced by the cache efficiency i.e., cache locality ratio of *input_size/cache_size*. For fine-grained join performance evaluation, we configure the input size with 8 proportions, 25%, 50%,75%, 100%, 125%, 150%, 175% and 200% of different level of cache size (L1, L2, L3 cache slice and LLC).

Part 2 is to evaluate the candidate join algorithms with benchmark evaluations. Schemas in benchmarks integrated the database systematic optimizations e.g., normalization, denormalization, indexing, etc. The workload B in [1] with two 128M rows table join case may be optimized with denormalization like SSB to eliminate the costly big table join. Moreover, the dimension tables in benchmarks are commonly small and slowly increasing, this schema feature may dominate which join algorithm is adaptive to data warehousing workloads. To the best of our knowledge such a study has not been conducted. The experimental conclusions can give real-world advice for query optimizer design.

Part 3 demonstrates how we implement the join algorithms for different platforms with the representative multicore CPU, GPU and Phi platforms. The performance evaluations also discover how caching or simultaneous multi-threading mechanisms affect the join performance. The fine-grained join performance evaluation not only tell which join algorithm is better but also discover the join performance patterns on heterogeneous processor platforms with strengthen and weakness regions. These systematic evaluations can give valuable advice on how to choose proper join algorithm according to hardware characteristics and table size.

5.1 Dataset

We have used multiple datasets for comprehensive performance evaluations. For join performance evaluation, the size of the *S* table was set to 600 000 000 rows, whose scale is same to that of TPC-H and SSB with Scale Factor=100. SF=100 is the smallest dataset for OLAP benchmark in early time, so the join performance can be referenced for in-memory database benchmark evaluations. For CCJB testing, we set the size of the *R* table to proportions of cache size, such as 25%, 50%, 75%, …, 200% of L1, L2, L3 cache size. By varying *R* table size, we can measure how join performance is dominated by cache locality. For the benchmark evaluations, we set the sizes of *R* and *S* to the sizes of the dimension table and the fact table in SSB, TPC-H, TPC-DS respectively. By dataset size designs, we can evaluate join performance with industry application scenarios, so that the conclusions can be more valuable for database systematic designs and optimizations.

5.2 Update overhead

For OLAP workloads, updates on dimensional attributes (non-primary key attributes) have no effects on surrogate key index; insertions on dimension table and fact table have little effects on surrogate key index; deletions on fact table have no effects on surrogate key index, deletions on dimension table are rarely occurred for the foreign key reference constraints. Update on surrogate key index may be invoked for a long time when deletions on dimension are more than certain threshold.

For SSB and TPC-H, we simulate the updates on dimension tables with update ratio step 10% from 0 to 100%. By comparing the average update time and original AIR time, we find that small dimension is fast in updating and is sensitive to update ratios while big dimension spends more time on updating and less sensitive to update ratios. In SSB update tests, the dimension tables *date*, *supplier*, *part* and *customer* are average 1.765, 1.768, 1.532, and 1.519 times of original AIR processing time. In TPC-H update tests, the tables *customer*, *supplier*, *part*, *partsupp* and *orders* are average 1.441, 1.550, 1.402, 1.136 and 1.122 times of original AIR time. The update overhead of surrogate key index is lower than traditional hash joins.

As mentioned in section 3.3, we use TPC-DS to simulate logical surrogate key index performance. For 13 foreign key referencing tables (dimension tables and referenced *store_return* fact table), logical surrogate key index mechanism spends average 1.15% more time than physical surrogate key index oriented AIR. For detailed analysis, the average logical surrogate key index oriented building phase spends 149.87% more time than physical surrogate key index oriented building phase, this is because build phase of AIR occupies only average 0.82% execution time of whole AIR processing.

The OLAP workload features guarantee that updates for surrogate key index are not frequently invoked operations, the vector referencing oriented update operation and logical surrogate key index provide high update performance.

5.3 Platforms

The experiments were performed on a DELL Power Edge R730 server with two Intel Xeon E5-2650 v3 @ 2.30GHz CPUs and 512 GB DDR3 main memory. Each CPU has 10 cores and 20 physical threads. The OS is CentOS, and the Linux kernel version is 2.6.32-431.el6.x86_64. The gcc compiler version is 4.4.7. The server is equipped with an Intel Xeon Phi 5110P coprocessor, which has 60 cores and 240 threads. The Phi processor has 8 GB on-board memory with bandwidth of 320 GB/s. Moreover, the server is also equipped with a NVIDIA Tesla K80 GPU, with 4992 cuda cores and 24 GB GDDR5 on-board memory. The bandwidth of GPU memory is around 480 GB/s.

We used the open source code from [2] for hash join performance testing. It contains a *hardware-conscious* join algorithm (PRO) and a *hardware-oblivious* join algorithm (NPO). AIR algorithm in [12] is easily to be implemented inside the hash join source code, we just replace the shared hash table of NPO with shared vector, and the hash probing is also replaced by address probing on vector by *key* of *S* table. We disable *knuth_shuffle* function for generating table *R* to guarantee the key to be surrogate key index. The performance is evaluated with *cycles-per-tuple* as [1] and [2]. We configure NUM_PASSES 2 and NUM_RADIX_BITS 14 parameters for PRO which experimentally yields the best cache residency for our CPU hardware configuration, and we run the NPO

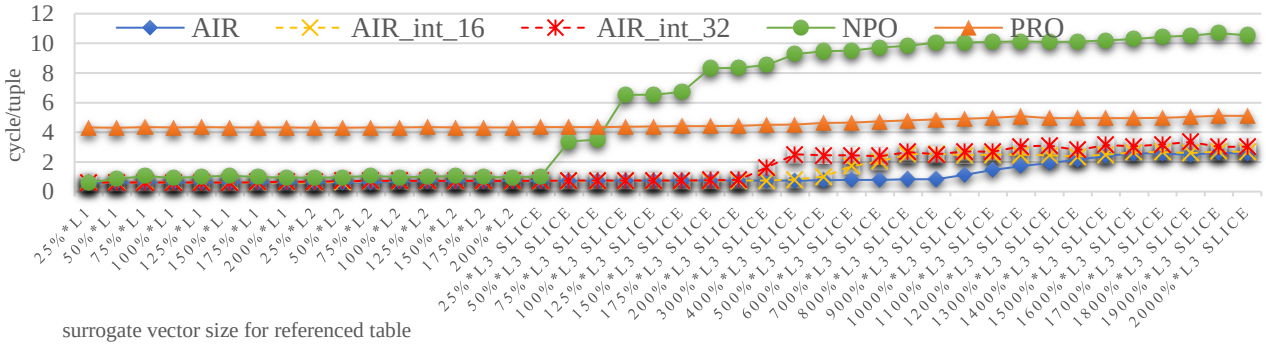


Figure 15. Join algorithms performance of AIR, NPO, PRO with different vector sizes and widths

and PRO algorithm on CPU platform to verify that the performance is similar to the results of [1] to guarantee the optimal configurations. Note that PRO is obviously low performance than NPO for joins with small hash tables, we perform the complete test to discover exactly when PRO begins to outperform NPO.

For Phi performance evaluation, we compile the source code with icc for Phi version. The Phi version of NPO and PRO algorithms uses the open source code from [6]. As CPU and Phi have different frequency, we use *ns-per-tuple* to evaluate performance as [6]. For GPU tests, we use AIR algorithm to evaluate join performance because AIR can be considered as *hardware-oblivious* and *schema-conscious* join algorithm, AIR is also the tailored NPO join algorithm with PRIMARY KEY AUTOINCREMENT constraint, compression on *payload*, and configuring BUCKET_SIZE=1. Compared with NPO, the simple structure and address probing also enable AIR to be *platform-oblivious* design for different characteristic processors. The AIR algorithm is programmed with cuda language, table *R*, *S* and vector are implemented with *int* type arrays, we use *pinned* memory to allocate GPU memory for high performance. In experiments, we configure BLOCK_NUM 16 and THREAD_NUM 512 parameters for optimized cuda *kernel* functions, the join results are merged in shared memory of GPU.

5.4 Cache-Conscious Join Benchmark

For the experiments with *cache-conscious* join benchmark, we design a group of join scripts with different *R* sizes, in which vector sizes in AIR are set in proportion with the L1(32KB), L2(256KB), and L3 cache slice(2.5MB) size, the maximal 2000% L3 slice denotes vector size is 2 times of LLC size. NPO and PRO use hash table to store tuples in table *R*. The hash table size is larger than surrogate vector used by AIR.

Figure 15 gives performance curves of the AIR, NPO and PRO join algorithms. NPO's performance curve can be clearly divided into 3 stages: when the hash table size is smaller than the cache size (L1, L2, L3 cache slice), the number of CPU cycles consumed by each tuple is pretty low; when the hash table size is larger than a L3 cache slice (2.5MB), the number of CPU *cycles-per-tuple* begins to rise as remote cache access is required; when the hash table is larger than the entire L3 cache, each hash probing may involve one or more cache misses, resulting in a large number of CPU *cycles-per-tuple*. The *hardware-oblivious* NPO algorithm is sensitive to hash table size, and it can automatically achieve good performance with small size of *R* table. PRO's performance curve in Figure 15 is almost constant. The size of the *R* relation does not affect the cache efficiency severely, due to the employment of data partitioning can guarantee in-cache hash building and probing. Therefore, the *hardware-conscious* PRO algorithm is not sensitive to table size. The AIR algorithm can be considered as a new *hardware-oblivious* algorithm with *schema-conscious* design. The performance curve of AIR algorithm in Figure 15 appears quite like NPO but much flatter. AIR is also sensitive to the surrogate vector size when the *R* table is small. When the *R* table is large, the number of *cycles-per-tuple* rises more slowly than NPO. Nevertheless, its overall performance is always better than that of NPO and PRO.

The size of the surrogate vector is an important factor to dominate the performance of AIR. It is usually in proportion with the width of the surrogate key. Figure 15 also shows the performance curves of AIR algorithm where we set the type of the surrogate key to *int*_8, *int*_16 and *int*_32 respectively. For wider surrogate vector, the rising up stage comes earlier, then *cycles-per-tuple* value still remains slowly increasing. When surrogate vector size exceeds L3 cache slice, AIR with different vector width has similar performance.

The width of surrogate vector defines the cardinality of projected attributes. For example, the cardinality of grouping attributes in one dimension table is commonly lower than 255 in SSB and TPC-H, for example, the maximal grouping size of SSB with 4 dimensions is 800. With compression technique, we can reduce surrogate key size to *int*_8. [12] proposed a systematic in-memory OLAP approach with AIR, and had proved that the *int*_8 vector can satisfy the whole SSB queries. In the following experiments, we set surrogate vector type to *int*_8 as default, and the AIR algorithm can represent the ultimate performance of *hardware-oblivious* join algorithm with *schema-conscious* optimizations to simplify hash table and hash probing overhead including compression and surrogate key index.

AIR algorithm employs one-to-one map between dimension table and surrogate key vector, queries with different selectivity use the fixed length vector with different distributions of *NULL* cells. Hash join builds hash table for tuples filtered by predicates, highly selective query on big table may produce small hash table which can achieve high performance with NPO algorithm.

For query optimizer, the performance curves in Figure 15 can be used as an optimization rule to choose the best join algorithm for given query. Let *V* denote vector size, let *s* denote selectivity, for a query with selectivity *s*, we can get *y-axis* values by *x-axis* value *V* and *V***s* on AIR and NPO performance curves. By comparing the *y-axis* values, we can choose the best join algorithm. A rough observation from Figure 15 is that if vector size *V* exceeds 1000% L3 cache slice ($25*2^{20}$ rows) and *V***s* is lower than 25% L3 cache slice ($0.625*2^{20}$ rows), i.e. selectivity is lower than 2.5%, we should choose NPO algorithm opposite to AIR as better choice.

The performance curves of *cache-conscious* join benchmark discovered how join performance is influenced by cache locality ratio. The join performance can also be modeled as a smooth curve dominated by several key performance points, and the performance curve can be used to predicate join performance instead of traditional evaluation oriented cost model.

As the conclusion of this subsection, within $50*2^{20}$-row tables, AIR algorithm outperforms PRO algorithm. NPO algorithm is preferred for tables larger than $25*2^{20}$ rows and with very low selectivity (lower than 2.5%), AIR algorithm is preferred for queries either with high selectivity (larger than 2.5%) or with small table sizes (smaller than $50*2^{20}$ rows).

5.5 Benchmark evaluation

We used workload A and workload B from [2] to evaluate the performance of join. In workload A, relation *R* has $16*2^{20}$ tuples and relation *S* has $256*2^{20}$ tuples to simulate a big table joining with a small table. In workload B, both relation *R* and *S* have $128*10^6$ tuples to simulate two big table join. Considering the use of surrogate key on dimension tables and the common application of dictionary compression, we set the *key* and *payload* attributes of the *R* and *S* table to *int*_32, resulting in a 4-byte/4-byte tuple structure.

Workload A conducts join between a large table and a small table. In the experiments, we kept increasing sizes of *R* and *S* with the same proportion, until reaching the maximal surrogate key value of

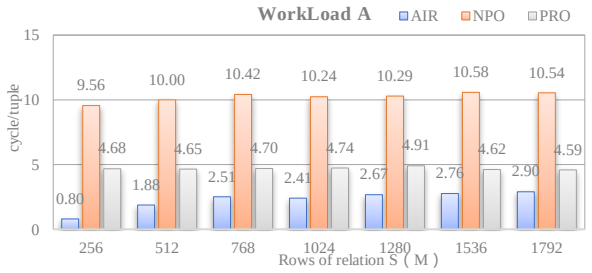


Figure 16. Join performance for AIR, NPO and PRO with workload A

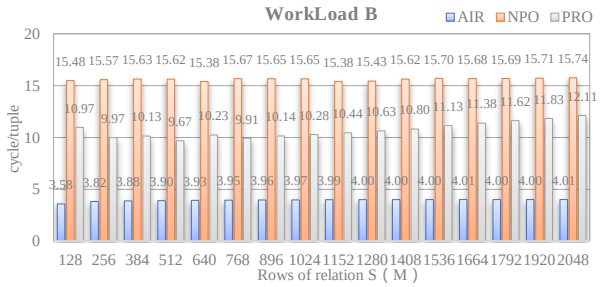


Figure 17. Join performance for AIR, NPO and PRO with workload B

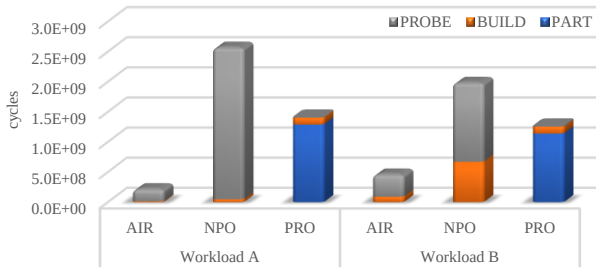


Figure 18. Breakdown of join execution time

R. As shown in Figure 16, NPO and PRO show the stable performance, while AIR slowly decreases performance as the size of *R* increases for surrogate vector size becomes larger than LLC size. When the dataset is large enough, the performance of AIR, NPO and PRO all become stable, as caching is no longer effective to AIR and NPO at this stage. According to the metric of *cycles-per-tuple*, NPO is about 2.3 and 3.6 times as costly as PRO and AIR.

Workload B conducts join between two large tables, where *R* and *S* both contain 128 million rows. We kept increasing the sizes of *R* and *S* until the surrogate key of *R* reaches the maximal value. The experimental results in Figure 17 show that NPO and AIR perform constantly as the sizes of *R* and *S* increase, while PRO's performance deteriorates slowly for the increasing overhead of partitioning. For Workload B, NPO is about 1.3 and 3.9 times as costly as PRO and AIR. The performance gap between NPO and PRO shrinks in this workload, while the performance gap between NPO and AIR grows wider.

Table 2. Join benchmark for SSB

SSB	SF=100		Cycles/tuple		
Tables	**Dimension**	**Fact Table**	**AIR**	**NPO**	**PRO**
date	2555	600000000	0.614	0.704	4.2459
supplier	200000	600000000	0.609	1.0634	4.3465
part	1528771	600000000	0.7283	3.2204	4.3372
customer	3000000	600000000	0.7367	6.4965	4.3479
	SF=200		**AIR**	**NPO**	**PRO**
date	2555	1200000000	0.6096	0.7027	4.313
supplier	400000	1200000000	0.7005	1.0438	4.3342
part	1728771	1200000000	0.7274	3.2871	4.3297
customer	6000000	1200000000	0.7387	8.3506	4.3763
	SF=300		**AIR**	**NPO**	**PRO**
date	2555	1800000000	0.6107	0.6967	4.1791
supplier	600000	1800000000	0.7109	0.9238	4.2186
part	1845764	1800000000	0.7281	3.1701	4.2174
customer	9000000	1800000000	0.7448	9.1763	4.2557

Table 3. Join benchmark for TPC-H

TPC-H	SF=100		Cycles/tuple		
Tables	**Dimension**	**Fact Table**	**AIR**	**NPO**	**PRO**
customer	15000000	150000000	0.8353	9.8735	4.9176
supplier	1000000	600000000	0.7218	0.9672	4.3393
part	20000000	600000000	0.7992	9.7736	4.5618
partsupp	80000000	600000000	2.5841	10.6461	5.3588
orders	150000000	600000000	3.0952	11.2908	5.641
	SF=200		**AIR**	**NPO**	**PRO**
customer	30000000	300000000	2.5102	10.3537	4.8683
supplier	2000000	1200000000	0.7367	3.3188	4.3356
part	40000000	1200000000	2.4603	9.9922	4.5782
partsupp	160000000	1200000000	3.1608	10.7107	5.4246
orders	300000000	1200000000	3.3644	11.3414	5.9213
	SF=300		**AIR**	**NPO**	**PRO**
customer	45000000	450000000	2.6388	10.5868	4.8345
supplier	3000000	1800000000	0.732	6.4765	4.3491
part	60000000	1800000000	2.509	10.2517	4.4295
PARTSUPP	240000000	1800000000	3.1557	10.9847	5.3094
orders	450000000	1800000000	3.3745	11.6566	5.8831

To get a deeper insight, we further measure the execution time of the different stages of the join algorithms. Figure 18 shows the breakdown of the execution time of AIR, NPO and PRO. We can see that PRO achieves the shortest probing time at the cost of data partitioning. AIR's probing time is higher than that of PRO but much lower than that of NPO, as it can achieve high code efficiency by leveraging the surrogate key based index, the overhead of vector generation for AIR is also lower than the overhead of building shared hash table in NPO. NPO's hash probing is slow, due to both cache misses and the cost of hash key processing overhead.

Furthermore, benchmarks like SSB, TPC-H and TPC-DS are used to measure the performance of data warehouse workloads. We

Table 4. Join benchmark for TPC-DS

TPC-DS	SF=100		Cycles/tuple		
Tables	**Dimension**	**Fact Table**	**AIR**	**NPO**	**PRO**
reason	55	28795080	0.6683	0.692	4.605
store	402	287997024	0.6241	0.6394	3.9484
promotion	1000	287997024	0.6203	0.6075	4.305
household_ dmographics	7200	287997024	0.6142	0.6384	4.4055
date_dim	73049	287997024	0.6176	0.9132	4.3674
time_dim	86400	287997024	0.6175	1.1005	4.3518
item	204000	287997024	0.6117	1.0413	4.3672
customer_ address	1000000	287997024	0.7237	0.9866	4.3885
customer_ dmographics	1920800	287997024	0.7351	3.2517	4.3886
customer	2000000	287997024	0.7348	3.3546	4.4287
store_returns	28795080	287997024	1.4666	10.3612	4.8286
	SF=300		**AIR**	**NPO**	**PRO**
reason	60	86393244	0.6334	0.6292	4.1749
store	804	864001869	0.6106	0.6352	3.8877
promotion	1300	864001869	0.6121	0.6871	3.9013
household_ dmographics	7200	864001869	0.6103	0.6322	4.3374
date_dim	73049	864001869	0.6117	0.9083	4.3125
time_dim	86400	864001869	0.6125	1.0972	4.3295
item	264000	864001869	0.6553	0.8901	4.3133
customer_ address	2500000	864001869	0.7366	6.5171	4.3529
customer_ dmographics	1920800	864001869	0.7289	3.4382	4.7154
customer	5000000	864001869	0.7439	8.248	4.3391
store_returns	86393244	864001869	2.7531	10.613	4.798

measure the performance of PK – FK joins between dimension table and fact table in these benchmarks. The experiments target at finding out which is the most adaptive join algorithm for the representative data warehouse workloads.

Our tests assigned row numbers to the relations *R* and *S* according to schemas of SSB, TPC-H and TPC-DS. We used the scale factors of 100, 200 and 300 in the tests.

Table 2 shows the results of AIR, NPO and PRO with the *cycles-per-tuple* metric on SSB. PRO still shows a constant performance at about 4.3 *cycles-per-tuple* for both small dimension table and large dimension table. NPO's performance depends on the size of the dimension table: it can achieve better performance than PRO with small dimension table, the minimal *cycles-per-tuple* is 0.704; only for the large *customer* table, it is slower than PRO. AIR outperforms both NPO and PRO in all the tests, and its maximal *cycles-per-tuple* is about 0.75. As the scale factor increases, PRO maintains constant performance while NPO's performance keeps decreasing. For the largest *customer* dimension table with the scale factor of 300, the surrogate vector of AIR is only 9 MB, which is smaller than the LLC size (25 MB), so that address probing on surrogate vector commonly occurs in cache to provide high performance.

Table 3 shows the join performance for TPC-H. The 3NF oriented TPC-H contains some big table joins such as *partsupp* ⋈ *lineitem*, *orders*⋈*lineitem*. Most of dimension tables in TPC-H are larger than their corresponding dimension tables in SSB. Therefore, NPO's performance is lower than PRO's in most of the tests. AIR is still superior to NPO and PRO in all the tests. For the test of the largest join, e.g., *orders*⋈*lineitem* (SF=300), NPO is 3.46 and 1.74 times as costly as AIR and PRO. In this test, although PRO shows a similar performance as AIR, it consumes 2 times as much memory as AIR, as it needs large intermediate space for data partitioning.

TPC-DS is more complex than SSB and TPC-H, as it contains more dimension tables with higher complexity. On the other hand, the scale of its schema follows most real-world cases, that dimension tables grow much slower than fact table, in contrast to the fixed proportion between the dimension and fact tables in TPC-H and SSB. For the joins in TPC-DS as shown in Table 4, NPO outperforms PRO in most cases, due to the skewed schema with small dimension tables. The vector size of the biggest dimension table *customer* is 5MB, so that AIR can perform efficient in-cache address probing to achieve high performance.

The industry benchmarks show schema perspective optimizations, e.g., denormalizing 3NF oriented TPC-H to star schema SSB to eliminate the costly big table join; designing multiple but small dimensions for fine-grained analytics, which make data warehouses reduce the dependency of big table join performance; using surrogate key to simplify foreign key join between fact table and dimension tables with address probing instead of key matching oriented hash probing. *Join optimization is not only algorithm optimizations but also database systematic design optimizations to make complex operation simple and efficient.*

5.6 Evaluation for skewed data

Skewed datasets commonly have negative influence on workload balance in multi-core parallel processing. For shared memory approaches, such as NPO using shared hash table and AIR using shared surrogate vector, skewed data distribution can improve the data locality in cache. Although skewed data result in skewed accessing on partitioned chunks, it imposes limited influence on PRO, which applies *hardware-conscious* in cache hash probing.

Figure 19 gives the performance curves for AIR, NPO and PRO on the SSB dataset (SF=100), with a *Zif* factor varying from 0 to 1.75. AIR and PRO are not influenced by *Zif* factor, while NPO is remarkably influenced. As shown in Table 2, the maximal surrogate vector size of SSB (SF=100) is 3 MB (for the *customer* table). It can always fit in caches, so the skewed data distribution has little influence for in cache vector addressing. PRO intentionally partitions a big table into cache-fit small chunks, so that the building and hash probing are all processed in cache. Therefore, skewed data accesses also have little influence on PRO. In contrast, NPO is sensitive to skewness. When the shared hash table of NPO is small enough to be held in cache, e.g., the *date* table in Figure 19, the *Zif* factor also has little influence for NPO. When the hash table cannot be accommodated by the cache, skewness becomes beneficial to cache efficiency, e.g., the *supplier*, *part* and *customer* tables in Figure 19, the skewed distribution enables the "hot" dataset to be frequently accessed to improve the cache locality.

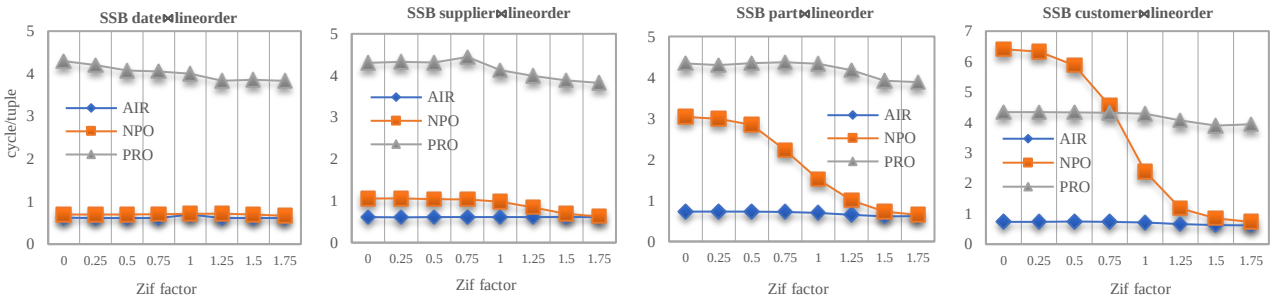


Figure 19. Join performance for skewed SSB data (SF=100)

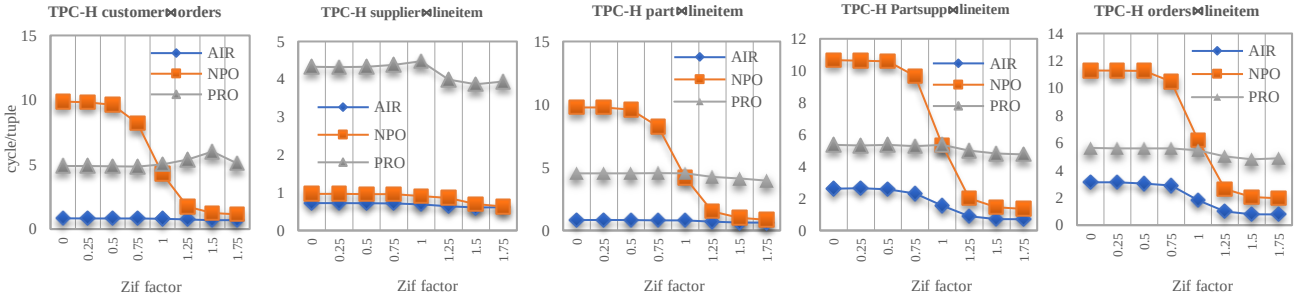


Figure 20. Join performance for skewed TPC-H data (SF=100)

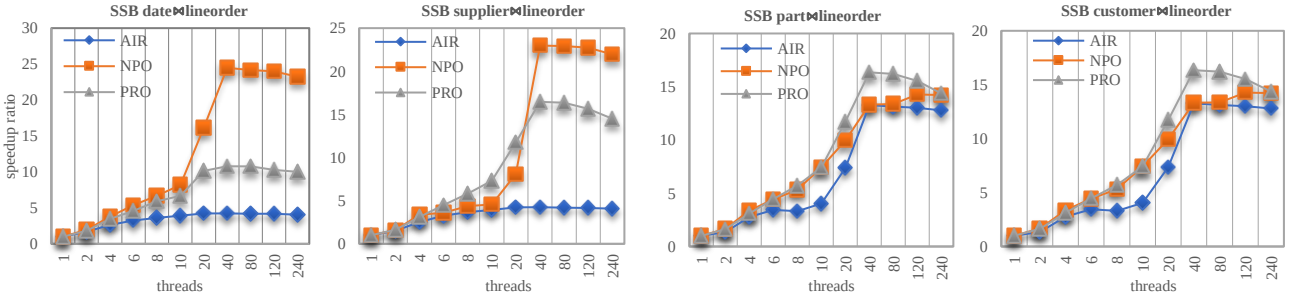


Figure 21. Parallel speedup ratio for AIR, NPO and PRO with SSB dataset (SF=100)

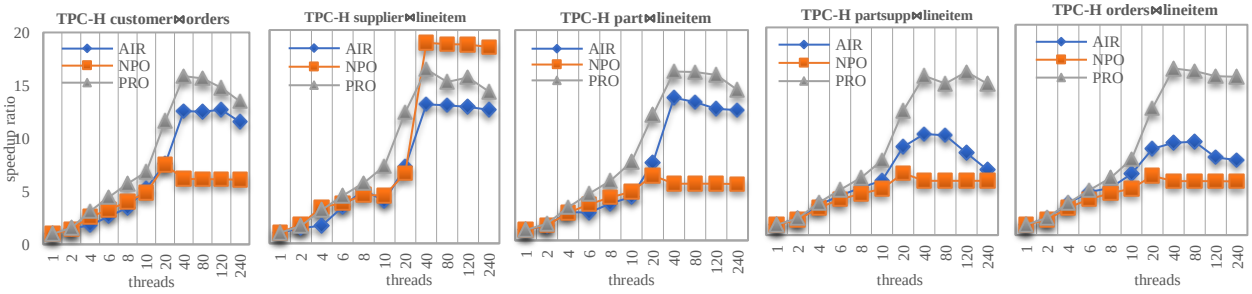


Figure 22. Parallel speedup ratio for AIR, NPO and PRO with TPC-H dataset (SF=100)

On TPC-H, we get the similar results. The curves of AIR and NPO drop remarkably as *Zif* factor increases for big tables, while PRO's curve drops slowly as shown in Figure 20. For NPO and PRO algorithms, when *Zif* factor is larger than 1, *hardware-oblivious* NPO outperforms *hardware-conscious* PRO. AIR is always superior to NPO and PRO, it also benefits from skewed dataset for self-adaptive higher cache locality.

Although hardware-conscious PRO tuning algorithm to hardware characteristics to improve performance, it also excessively optimizes join algorithm for skewed data and small table scenarios, missing the self-adaptive feature like hardware-oblivious algorithms such as NPO and AIR.

5.7 Evaluation for parallel scalability

The speedup ratio metric measures the parallel scalability of join algorithm. The concerned hardware factors mainly include number

of memory channels, cores and threads. The concerned software factors mainly include the size of the private dataset for each thread that can cause LLC contention and the CPU cycle consumption. Our testing platform was equipped with 4-channel DDR3 memory and two 10-core CPUs (each core supports 2 threads).

Figure 21 shows the speedup ratio charts for SSB dataset (SF=100). The *x-axis* represents number of threads, due to the 10-core and 20-thread architecture, we increase threads by step=2 within 10 threads, then use 20 threads and times of 20 threads to guarantee workload to be assigned to all the available threads equally.

We can see that in the joins with small *date* and *supplier* tables, the minimal speedup ratio is about 4 (memory channel number) for AIR, and the maximal speedup ratio is around 25 (slightly more than core number) for NPO. In joins with large *part* and *customer* tables, the maximal speedup ratio of three join algorithms is around 15 (slightly lower than core number) for PRO.

The size of the surrogate vector for the *date* table is 2.6 KB and that for the *supplier* table is 200 KB. Therefore, the surrogate vector address probing of AIR mostly occurs within the private L1 or L2 cache. As a result, the AIR join can be as efficient as parallel table scan. Thus, its maximal speedup ratio approaches 4, the number of memory channels. When the dimension table is small, NPO can achieve a high speedup ratio as its hash table can fit in the cache. For big dimension tables, PRO can achieve higher speedup ratio than NPO and AIR, due to it partitions the tables and joins each partition pair in parallel. Basically, the partitioning and probing phases of PRO are adaptive to be parallelized, and its contention for shared cache is lower than that of NPO and AIR.

Figure 22 shows the speedup ratio charts of TPC-H. For joins with small *supplier* table, NPO outperforms PRO and AIR. However, for big table joins, NPO achieves the lowest speedup ratio of around 5, while PRO and AIR usually achieve the maximal speedup ratio of around 15. PRO's speedup ratio is higher than AIR on big table.

As a summary, PRO uses a divide-and-conquer strategy to divide big table into small chunks and processes small chunks in parallel. Its parallel scalability is constant and high, regardless of the table size. NPO and AIR perform accesses on shared hash table or shared surrogate vector. If shared hash table or surrogate vector is smaller than LLC, parallel threads with shared LLC accessing achieves high scalability; if shared hash table or surrogate vector size exceeds LLC size, parallel threads with shared data accesses produce cache misses to contend for LLC resources. Therefore, their scalability is not as good as that of PRO.

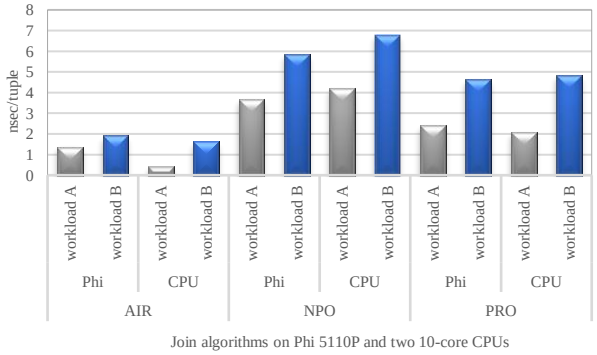


Figure 23. Join performance for AIR, NPO and PRO on CPU and Phi

5.8 Evaluation for heterogeneous platforms

The AIR algorithm uses arrays as the main computing instruments. Therefore, it is easily to be implemented on the emerging accelerator coprocessors, such as Xeon Phi, GPU and FPGA. In this section, we measure the performance of the join algorithms on two leading coprocessor platforms.

We used the open source from [6] as the Phi version NPO and PRO algorithms, and compiled the open source from [1] with embed AIR algorithm to Phi version for the tests.

Figure 23 shows the results of measuring AIR, NPO and PRO's performance on CPU and Phi in Workload A and B of [6]. We used two 10-core E5-2650 v3 CPUs with 40 threads and one Phi 5110P coprocessor with 60 cores and 240 threads. NPO's performance on Phi is slightly higher than its performance on CPU. PRO's performance on Phi is slightly lower than its performance on CPU. We can say that one Xeon Phi is comparable to two Xeon CPUs for both NPO and PRO. In contrast, AIR's performance on CPU seems remarkably superior to its performance on Phi with Workload A.

Figure 24 shows the results of join performance with SSB and TPC-H benchmarks. Frist of all, PRO fails all the tests with SF=100 dataset for the double memory consumption which exceeds the size of device memory of Phi, NPO fails for large table join in TPC-H, and AIR passed all the tests for its efficient vector store which is

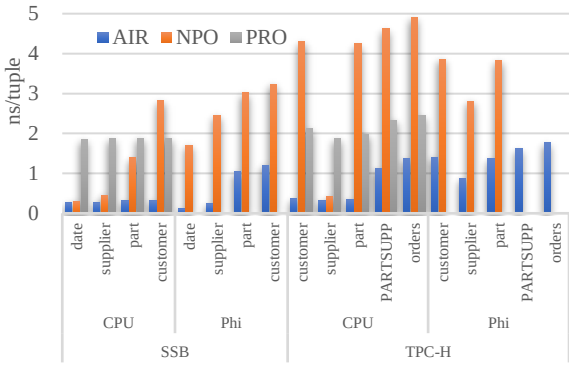


Figure 24. Join performance with SSB and TPC-H

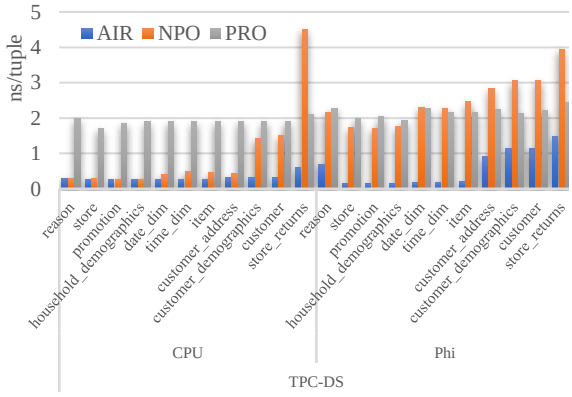


Figure 25. Join performance with TPC-DS

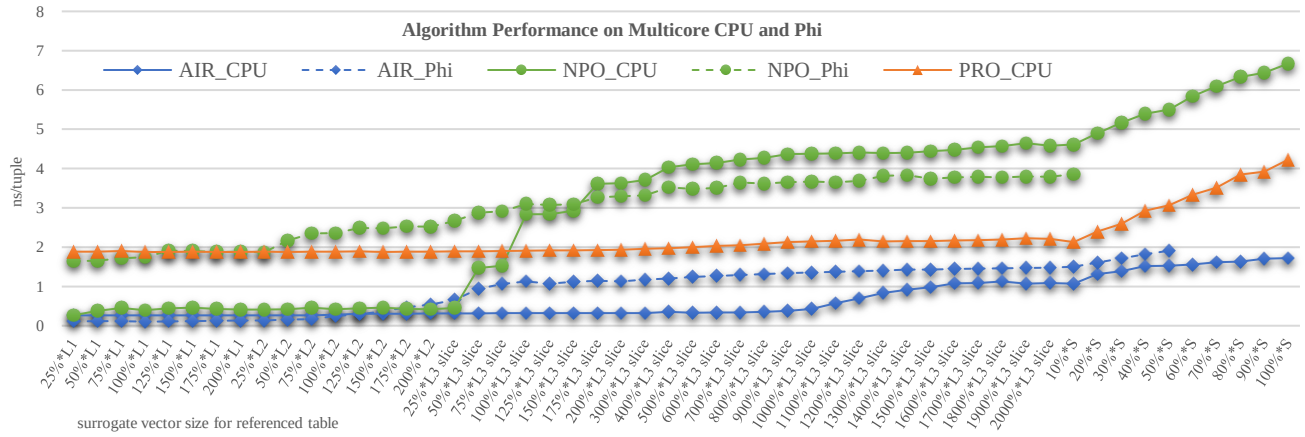


Figure 26. Join algorithms on CPU and Phi

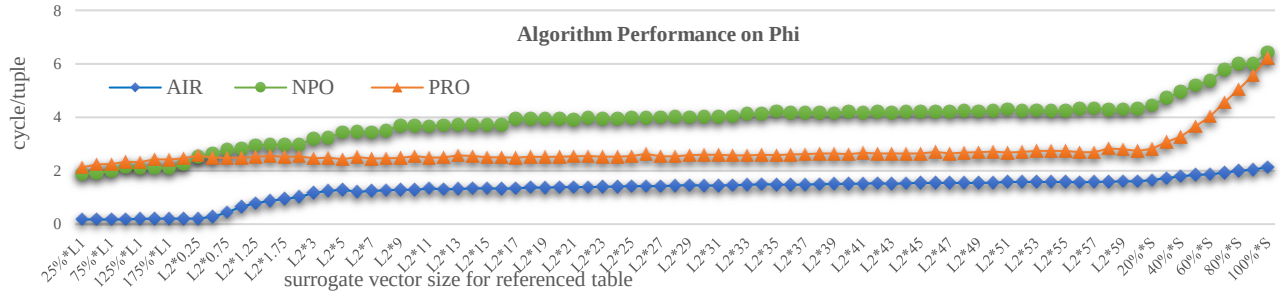


Figure 27. Join algorithms on CPU and Phi with small dataset

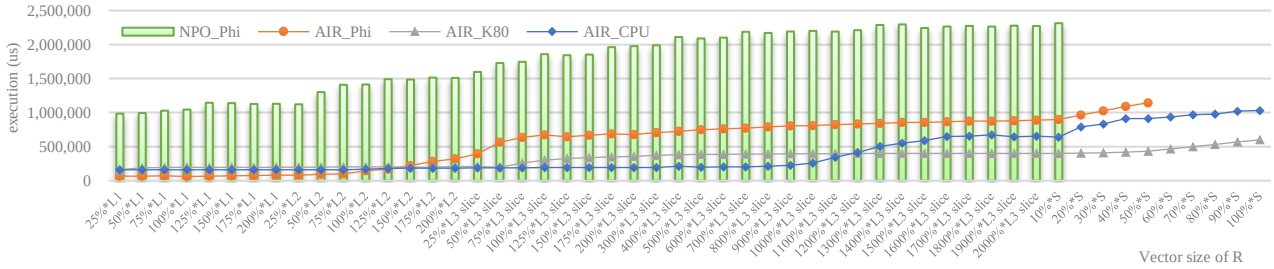


Figure 28. Join algorithms on CPU, Xeon Phi and K80 GPU

enabled by surrogate key index and compression techniques of database. From the perspective of memory efficiency, AIR outperforms NPO and PRO. Considering the performance on CPU and Phi, we see some interesting results. AIR and NPO perform better on CPU than on Phi with SSB, the main reason lies in that the small dimension size and larger LLC size on CPU produce higher cache locality than on Phi. With TPC-H, except the small *supplier* table, the join performance with large table performs better on Phi than on CPU, the main reason lies in that *simultaneous multi-threading* mechanism of Phi works when hash table size exceeds the cache size. For AIR algorithm, AIR on Phi outperforms on CPU with *date* and *supplier* tables in SSB because the vector size is lower than L2 cache and more threads of Phi produce higher performance; AIR on CPU outperforms on Phi for other tables with vector size larger than L2 of Phi, the larger L3 cache on CPU produces higher cache locality.

Figure 25 shows the join performance with TPC-DS on CPU and Phi. PRO still illustrates the constant performance both on CPU and Phi; NPO's performance differs with different size of dimension tables. For small dimension tables, NPO on CPU outperforms on Phi, for large dimension tables, NPO on Phi outperforms on CPU. AIR on CPU outperforms on Phi mainly for its cache fit vector size.

For a deeper insight, we repeated the *cache-conscious* join benchmark in section 4.3 on Phi. Phi's L2 cache is of 512 KB, while CPU's L2 cache size is 256 KB. CPU's L3 cache is 25 MB, while Phi has no L3 cache but a 30 MB shared L2 cache ring. The axis "Vector size of *R*" represents surrogate vector (*int*_8) size, in proportion with the sizes of the cache at different levels. For example, "25% L1 cache" represents that the surrogate vector size is 0.25*32 KB=8KB, the row number is $8*2^{10}$.

Figure 26 shows the performance curves of AIR, NPO and PRO on CPU and Phi platforms. For small tables with surrogate vector size lower than L2 cache (256 KB), AIR on Phi is faster than AIR on CPU for massive parallel threads. When surrogate vector size is larger than 150% of L2 (384 KB, 75% of Phi's 512 KB L2 cache), CPU's performance begins to outperform Phi's performance due to

the large L3 cache slice size (2.5 MB). For small *R* table sizes, AIR and NPO both have good performance, while they are quite different on Phi. For a small *R* table size whose size is lower than the L3 cache slice (100% L3 cache slice on Figure 26), NPO on CPU outperforms NPO on Phi, the major reason may be the performance of Phi's P54C core is lower than CPU core for either low frequency or missing advanced instructions like *out-of-order execution*. For a large *R* table, NPO on Phi outperforms NPO on CPU, due to its high parallelism. NPO on Phi starts to outperform it on CPU when row number exceeds 3.75M (150%*L3 cache slice), and NPO's performance is lower than PRO on CPU for most cases.

Due to the limited on-board memory of Phi (8GB), PRO with doubled memory consumption cannot be conducted. As shown in Figure 26, AIR can support larger table size than NPO, as its surrogate vector consumes less space than the hash table of NPO. PRO completely fails the test, as data partitioning consumes twice as much space as the data itself. *Therefore, PRO is not a good choice when memory space is scarce*.

Moreover, we reduced the dataset to SF=20 to enable all the join algorithms to be performed on Phi. Figure 27 shows the join performance with *cache-conscious* join benchmark, we increase the vector size up to shared L2 cache (60* 512KB). The performance curves show that NPO only outperforms PRO within small input cases about lower than 64K rows, PRO outperforms NPO for large input at the cost of double device memory consumption. AIR is about half of the execution cost of PRO's, and is influenced by vector size compared with L1 cache size, L2 private cache size, shared L2 cache size, and slowly increases execution cost for large input cases. The join performance patterns on Phi are like patterns on CPU, caching is still the important mechanism to improve join performance. NPO improves performance on Phi than on CPU by more threads, but still runs slower than PRO.

Compared with x86 architecture CPU and Phi, GPU has different architecture. The cache is very small than CPU (128KB shared memory as L1 cache for 192 cuda cores within one SMX, 1.5 MB L2 cache for entire GPU), the small cache is not adaptive to parallel radix partitioning based hash join algorithm of PRO. The *hardware-oblivious* NPO can improve hash probing performance on GPU with the SIMT mechanism for thousands of parallel threads, but the hash table structure, latch mechanism for building hash table, branching processing for hash tuple matching must be carefully tuned for GPU hardware features. AIR simplifies the hash table structure as surrogate vector without pointer, the hash probing is also simplified as address probing to single vector address, so that AIR is adaptive to be implemented with cuda to run on GPU.

We implemented the AIR algorithm on NVIDIA K80 GPU, AIR uses the fixed length surrogate vector for addressing probing, this design is adaptive to GPU *cudamalloc* function to pre-allocate device memory on GPU. We only measured the *kernel* execution time of AIR on the GPU. As K80 has 24 GB device memory, testing with various row numbers can all be executed on K80. Figure 28 gives the join execution time (*us*) for AIR on CPU, AIR on Phi, NPO on Phi and AIR on K80. We can see the different performance curves of AIR on three processors.

AIR on CPU performs well as vector size of *R* is lower than LLC size (1000%* L3 slice). AIR on Phi outperforms the other two as vector size of *R* is lower than 150% L2 cache size (75% of Phi L2 cache, note that L2 cache size of Phi is 512KB instead of 256KB of CPU) for the massive parallel threads, when vector size exceeds L2 cache (200%*L2, 512KB), the performance begins to decrease significantly. While AIR on K80 is relatively flat because K80 doesn't rely on caching mechanism instead of SIMT mechanism to overlap memory access latency.

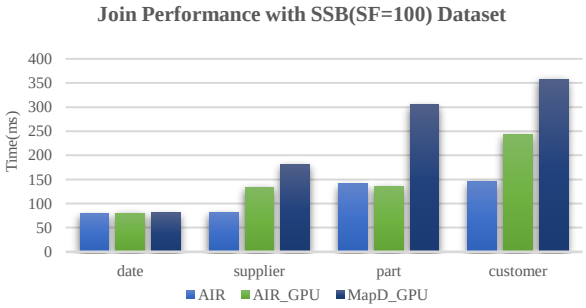


Figure 29. AIR performance with MapD join

This chart discovers performance patterns of caching and simultaneous multi-threading mechanisms. *Caching still plays the important role for improving join performance, the LLC size dominates the input relation size with high performance.* Nowadays CPUs have limited LLC size for the limited core number, the new KNL processor's 16GB HBM cache can remarkably improve join performance with big input relation. The first-generation Phi processor is less efficient for shared L2 cache with dual ring bus architecture, join on Phi has the highest performance but limited with small input size (lower than L2 cache not the shared L2 cache). The join performance can be improved on second-generation KNL Phi processor with tile architecture. *For x86 processors such as CPU and Phi, the join performance strengthen region is cache fit input size, the weakness region is large input exceeding LLC.* GPU has the input size oblivious performance pattern, join performance doesn't influenced much by different input size. For large input size, the performance is still high. The GPU join performance strengthen region is large input, the weakness region is cache fit small input size. *We can draw the conclusion that x86 processors is adaptive to small table join with caching mechanism to improve data locality and GPU is adaptive to big table join with SIMT to overlap memory access latency.*

As the performance curve of GPU is almost flat, less performance gap is left for PRO to further improve join performance, we neglect developing GPU based NPO and PRO baseline algorithms in this paper, and we will fill this blank in our future work.

Additionally, we used GPU database MapD for baseline GPU join performance. In Figure 29, we can see that AIR on CPU outperforms AIR on GPU for SSB joins, MapD join has similar performance to AIR on GPU. The performance gap can be considered as surrogate key index performance gains against traditional join performance.

Different join algorithms have special patterns under different workloads. Table 5 summaries the performance patterns of NPO, AIR and PRO algorithms with different benchmarks on different platforms, these patterns describe the strength and weakness regions of each join algorithm and can provide valuable information for query optimizer design.

Table 5. Join performance pattern comparison (SF=100)

Algorithms		CCJB	SSB	TPC-H	TPC-DS
NPO	CPU	H: <512K row L: >2.5M row	H: 3 dims L: 1 big dim	H: 1 dim L: 4 dims	H: 8 dims M:2 L:1
	Phi	M: <128K row	H:2 M:1 L:1	H: 1 M: 4	H:8 M:8 L:6
	GPU	N/A	N/A	N/A	N/A
AIR	CPU	H: <22.5M row M: >25M row	H: all	H: all	H: all
	Phi	H: <256K row M: >512K row	H:2 M:2	H:1 M:4	H:19 M:3
	GPU	H:>30M row M:<30M row	M	H	M
PRO	CPU	M: constant	M: constant	M: constant	M: constant
	Phi	Fail	Fail	Fail	Fail
	GPU	N/A	N/A	N/A	N/A

** H: high, M: moderate, L: low of three join candidates*

From table 5, we have the following observations:

- The performance of *hardware-oblivious* join algorithms (NPO and AIR) is dominated by cache locality ratio, either reducing hash table size (simplifying hash table as efficient vector in AIR) or enlarging LLC size (employing KNL processor with large memory-side cache) can improve join performance, performance of different joins differs little within cache size input relations;
- The performance of *hardware-conscious* join algorithm provides constant performance for different workloads, the strengthen region is big table join, the weakness region is cache-fit small table join, the performance is obtained at the cost of doubling memory consumption in partitioning phase;
- With *schema-conscious* design, NPO is simplified as AIR by transforming hash probing to address probing with PRIMARY KEY AUTOINCREMENT constraint or surrogate key index, the compressed vector for data warehousing workloads minimizes vector size to be held in cache even for big table, the *schema-conscious* and *hardware-oblivious* AIR algorithm outperforms *hardware-conscious* PRO algorithm for both performance and memory efficiency;
- Database systematic optimization is another important perspective for improving join performance, and the well-designed schemas reduce the costly big table joins in workloads, and enable the simple *hardware-oblivious* algorithms to be adaptive for OLAP workloads;
- Considering the complex optimization techniques of hash join [11], the *hardware-oblivious* NPO algorithm is not the best candidate for heterogenous processors, AIR algorithm eliminates the optimization requirements by removing complex structure and processing by database systematic optimizations to enable AIR to be *platform-oblivious* design for heterogenous processors, the *hardware-oblivious* design should be upgraded to *platform-oblivious* design for the emerging high performance heterogenous platforms;
- The approaches that PRO is the best join algorithm for CPU platform commonly emphasize the performance perspective, considering the coming trend of integrating large HBM on chip, memory efficiency plays more important roles to enable processor processing more data.

6. RELATED WORK

The debate on whether *hardware-oblivious* or *hardware-conscious* join algorithm is better choice remains hot topic in recent years. It is a dilemma that how we consider hardware in designing algorithm, as a white box or as a black box? [1] first systematically evaluates the performance of *hardware-oblivious* no partitioning join and *hardware-conscious* radix join, and their conclusion was that *hardware-oblivious* join algorithm performs as well as *hardware-conscious* join algorithm with simple structure and algorithm design. No doubt, this conclusion can dramatically simplify the query engine design. While [2] proposed the completely opposite conclusion that *hardware-conscious* join algorithm beats *hardware-oblivious* join algorithm with further optimizations. Considering the SIMD optimization technique, [9] compared radix join and sort-merge join with deep SIMD optimizations, the radix join still wins. In database implementations, hash join algorithm is continuously improved, [17] proposed concise hash table to provide a linear probing with 100% fill factor to reduce hash table memory usage, the partitioning hash join is further optimized by only partitioning the build table and not partitioning the probe table to save memory consumption. These researches try to get the trade-off between high performance with fully partitioning and moderate performance with partial partitioning. And [18] developed the uniform testbed for all candidate join algorithms with same optimizations, datasets and workloads and try to give the final answer.

The debate is further pushed to the emerging hardware accelerator platforms. The join algorithms of GPU version [5], Phi version [6][7] and FPGA version [8] are proposed to verify whether *hardware-oblivious* or *hardware-conscious* join algorithm is adaptive to hardware accelerator architectures. With the continuously updated optimization techniques, the conclusion keeps refreshing. [11] discussed hashing methods with seven dimensions and 20 different combinations in total, the complexity of hashing methods makes query optimizer design even complex.

No partitioning join is a *hardware-oblivious* algorithm without concerning schema. Data warehouse is multidimensional model oriented opposite to relational mode. [12] proposed an array-store oriented AIR algorithm for foreign key join with PK-FK referencing constraint and OLAP *schema-conscious* design, the array engine gains high performance but limits the application scenarios. Array join in [18] has the similar consideration of [12]. AIR or array join algorithm can be considered as specialized NPO algorithm case when using PRIMARY KEY AUTOINCREMENT constraint or surrogate key index in dimension tables, and configuring BUCKET_SIZE=1. MonetDB use *fetchjoin* to directly access dense BAT with VOID, *invisible-join* [21] also illustrated how to use *positional lookup* (like *fetchjoin*) to access tuples from other column instead of column join. But the similar approaches are not employed for foreign key joins for lacking comprehensive mechanism to guarantee foreign key address mapping. Without the database systematic constraint mechanism, the *array join* can only be used for special cases.

Furthermore, the *schema-conscious* design can also simplify join algorithm on GPUs with simple array structure instead of complex hash table, and simple address probing instead of hash probing. Our work in this paper studied the performance of varied join algorithms by using the comprehensive data warehousing benchmarks.

7. CONCLUSIONS

In this paper, we re-evaluated the foreign key joins on main stream platforms. With *cache-conscious* join benchmark and industry benchmark testing, we discovered the join performance patterns with strength and weakness regions. For specified workload, the performance curve chart enable to accurately choose the best algorithm. The performance curve chart discovered the dependency of join performance, input size and cache size, and the cache locality ratio of *input_size*/*cache_size* can be used as threshold to predict the performance of join algorithms.

Based on the experimental results, the main conclusion is that the *hardware-oblivious* NPO join outperforms *hardware-conscious* PRO join in most benchmark workloads with small input table size and *hardware-conscious* PRO join prefers to big table joins. Moreover, the *hardware-oblivious* NPO join can be further optimized with *schema-conscious* and *OLAP workload customized* design and address probing operation, which outperforms *hardware-conscious* PRO for all the tests. Compared with hash table based NPO and PRO joins, AIR algorithm proves to be high performance for both small table and big table joins.

With heterogeneous processor perspective, a new performance issue is how to use the on-board memory efficiently. The *hardware-conscious* partition-based joins decrease the utilization of on-board HBM for the additional partitioning memory consumption. On the other hand, the *hardware-oblivious* joins can process large workloads for less intermediate memory consumption. The *hardware-oblivious* and *schema-conscious* AIR algorithm can provide both high performance and high memory efficiency joins on GPU and Phi processors. Moreover, AIR algorithm uses simple vector structure and address probing operation which enables it to be *platform-oblivious* implementation for heterogeneous processor platforms.

As a result, join algorithm should be designed and optimized based on schema features, hardware characteristics and platform heterogeneity to make joins efficient for workloads and the emerging heterogeneous architectures. As the future work, for data warehousing workloads, star-join is the fundamental operator for multidimensional operation, and we will exploit to optimize the join efficiency during pipelined star join.

Acknowledgements Yu Zhang is the corresponding author. This work is supported by Nature Science foundation of China Project No. 61732014,61772533 and Academy of Finland (310321).

8. REFERENCES

[1] Blanas, S., Li, Y., Patel, J. M. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *Proceedings of SIGMOD Conference*. ACM, New York, NY, pp. 37-48, Jun. 2011, doi:10.1145/1989323.1989328.

[2] Balkesen, C., Teubner, J., Alonso, G., and Ozsu, T. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *Proceedings of ICDE Conference*. pp. 362-373, Apr. 2013, doi:10.1109/ICDE.2013.6544839.

[3] Boncz, P. A., Kersten, M. L., Manegold, S. Breaking the memory wall in MonetDB. *Commun. ACM*. vol 51, no. 12, pp.77-85, Dec. 2008.doi: 10.1145/1409360.1409380

[4] Kaldewey, T., Lohman, G., Mueller, R., Volk, P. GPU join processing revisited. In *Proceedings of DaMoN Conference*. pp. 55-62, 2012.

[5] Yuan, Y., Lee, R., Zhang, X. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *PVLDB*, 6(10): 817-828 (2013).

[6] Jha, S., He, B., Lu, M., Cheng, X., Huynh, H P. Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach. *PVLDB*, 8(6): 642-653 (2015)

[7] Polychroniou, O., Raghavan, A., Ross, K.A. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of SIGMOD Conference*. pp. 1493-1508, May. 2015, doi:10.1145/2723372.2747645.

[8] Halstead, R.J., Absalyamov, I., Najjar, W.A., Tsotras, V. J. FPGA-based Multithreading for In-Memory Hash Joins. P. In *Proceedings of CIDR Conference*. Jan. 2015.

[9] Balkesen, C., Alonso, G., Teubner, Jens., Özsu, M. T. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *PVLDB* 7(1): 85-96 (2013)

[10] He, J., Lu, M., He, B. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. In *Proceedings of VLDB Conference*. vol. 6, no. 10, pp. 889–900, Aug. 2013.

[11] Richter, S., Alvarez, V., Dittrich, Jens. A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing. *PVLDB*, 9(3): 96-107 (2015)

[12] Zhang, Y., Zhou, X., Zhang, Y., Zhang, Y., Su, M., Wang, S. Virtual Denormalization via Array Index Reference for Main Memory OLAP. *IEEE Trans. Knowl. Data Eng*. 28(4): 1061-1074 (2016).

[13] Kemper, A., Neumann, T. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of ICDE Conference*. pp. 195-206, Apr. 2011, doi:10.1109/ICDE.2011.5767867.

[14] Sompolski, J., Zukowski, M., Boncz, P.A. Vectorization vs. compilation in query execution. In *Proceedings of DaMoN Conference*. 2011: 33-40

[15] Sikka, V., Färber, F., Lehner, W., Cha, S.K., Peh, T., Bornhövd, C. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *Proceedings of SIGMOD Conference*. 2012: 731-742

[16] Boncz, P. A., Zukowski, M., Nes, N. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of CIDR Conference*. 2005: 225-237

[17] Barber, R., Lohman, G.M., Pandis, I., et al. Memory-Efficient Hash Joins. In *Proceedings of VLDB Endowment*. 2015, Vol 8, No 4, 353-364

[18] Stefan, S., Xiao, C., Jens, D. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *Proceedings of SIGMOD Conference*. 2016: 1961-1976

[19] Avinash, S., Roger, G., Jesús, C., Ho-Seop, K., Krishna, V., Sundaram, C., Steven, H., Rajat, A., Yen-Chen, L. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro* 36(2): 34-46 (2016).

[20] Jack, D., Wen-Fu, K., Allen, K. L., Julius, M., Anirudha, R., Lihu, R., Efraim, R., Ahmad, Y., Adi, Y. Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake. *IEEE Micro* 37(2): 52-62 (2017)

[21] Daniel J. Abadi, Samuel Madden, Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of* SIGMOD Conference. 2008: 967-980

[22] Zhang Y, Wang S, Lu J. Improving performance by creating a native join-index for OLAP. Frontiers Comput. Sci. China 5(2): 236-249 (2011)